UB FILKOM
FAKULTAS ILMU KOMPUTER

Algoritma Brute Force

Oleh

Dr. Achmad Ridok, M.Kom dan Tim AKK

Bahasan

- Definisi Algoritma Brute Force (ABF)
- Karakteristik Algoritma Brute Force
- Contoh-contoh ABF
- Exhaustive Search (ES)
- Algoritma Heuristic

Definisi Brute Force

- *Brute force* : pendekatan yang lempang (*straightforward*) untuk memecahkan suatu masalah
- Biasanya didasarkan pada:
  - pernyataan masalah (*problem statement*)
  - definisi konsep yang dilibatkan.
- Algoritma *brute force* memecahkan masalah dengan
  - sangat sederhana,
  - langsung,
  - jelas (*obvious way*).
- *Just do it!*

Karakteristik Algoritma Brute Force

1. Algoritma *brute force* umumnya tidak "cerdas" dan tidak effective, karena ia membutuhkan jumlah komputasi yang besar dalam penyelesaiannya.

   Kata "force" mengindikasikan "tenaga" ketimbang "otak"

   Kadang-kadang algoritma *brute force* disebut juga algoritma naif (*naïve algorithm*).

2. Algoritma *brute force* lebih cocok untuk masalah yang berukuran kecil.

   Pertimbangannya:

   - sederhana,
   - implementasinya mudah

   Algoritma *brute force* sering digunakan sebagai basis pembanding dengan algoritma yang lebih effective.

Karakteristik Algoritma Brute Force

3. Meskipun bukan metode yang effective, hampir semua masalah dapat diselesaikan dengan algoritma *brute force.*

   Sukar menunjukkan masalah yang tidak dapat diselesaikan dengan metode *brute force*.

   Bahkan, ada masalah yang hanya dapat diselesaikan dengan metode *brute force*.

   Contoh: mencari elemen terbesar di dalam senarai.

Karakteristik Algoritma Brute Force

- **Ken Thompson (salah seorang penemu Unix) mengatakan:**

  "*When in doubt, use brute force*",

- **Percaya atau tidak, faktanya kernel Unix yang asli lebih menyukai algoritma yang sederhana dan kuat (*robust*) daripada algoritma yang "cerdas" tapi rapuh.**

Kekuatan dan Kelemahan ABT

Kekuatan:

1. **Dapat memecahkan hampir sebagian besar masalah**
2. **Sederhana dan mudah dimengerti.**
3. **Layak untuk beberapa masalah : pencarian, pengurutan, pencocokan *string*, perkalian matriks.**
4. **algoritma standard untuk tugas-tugas komputasi penjumlahan/perkalian *n* buah bilangan, penentuan elemen minimum atau maksimum di dalam tabel**

Kelemahan:

1. **Metode brute force jarang menghasilkan algoritma yang effective.**
2. **Beberapa algoritma brute force lambat sehingga tidak dapat diterima.**
3. **Tidak sekontruktif/sekreatif teknik pemecahan masalah lainnya.**

Contoh-contoh Algoritma Brute Force

1. Mencari elemen terbesar (terkecil)

- **Persoalan: Diberikan sebuah array yang beranggotakan *n* buah bilangan bulat (a_1, a_2, ..., a_n). Carilah elemen terbesar di dalam senarai tersebut.**
- **Algoritma *brute force*: bandingkan setiap elemen senarai untuk menemukan elemen terbesar**

```
CariElemenTerbesar(
        input A: array,
        output maks : integer)
Deklarasi
  k : integer
Algoritma:
  maks←a1
  for k←2 to n do
    if ak > maks then
      maks←ak
    endif
  endfor
```

Berapa Kompleksitasnya ?

2. *Sequential Search*

Persoalan:

Diberikan n buah bilangan bulat yang dinyatakan sebagai a1, a2, ..., an. Carilah apakah x terdapat di dalam himpunan bilangan bulat tersebut. Jika x ditemukan, maka lokasi (indeks) elemen yang bernilai x disimpan di dalam peubah idx. Jika x tidak terdapat di dalam himpunan tersebut, maka idx diisi dengan nilai 0.

```
PencarianBeruntun(
        input A : array,
        output idx : integer)
Deklarasi
  k : integer
Algoritma:
  k←1
  while (k < n) and (ak ≠ x) do
     k ← k + 1
  od
  if ak = x then
     idx←k
  else
     idx← 0 endif
  endif
```

Hitung Kompleksitasnya ?

3. Perkalian dua Matrik

Definisi:

Misalkan $C = A \times B$ dan elemen-elemen matrik dinyatakan sebagai c_{ij}, a_{ij}, dan b_{ij}

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Algoritma *brute force*: hitung setiap elemen hasil perkalian satu per satu, dengan cara mengalikan dua vektor yang panjangnya *n*.

```
PerkalianMatriks(
    input A, B : Matriks, n : integer,
    output C : Matriks)
Deklarasi
 i, j, k : integer
Algoritma
  for i  = 1 to n do
   for j =  to n do
     C[i,j] = 0
     for k = 1 to n do
       C[i,j] = C[i,j] + A[i,k]*B[k,j]
     endfor
   endfor
  endfor
```

Berapa Kompleksitasnya ?

4. CariFaktor

Menemukan semua faktor dari bilangan bulat *n* (selain dari 1 dan *n* itu sendiri).

Definisi: Bilangan bulat *a* adalah faktor dari bilangan bulat *b* jika *a* habis membagi *b*.

Algoritma *brute force*: bagi *n* dengan setiap *i* = 2, 3, ..., *n* – 1. Jika *n* habis membagi *i*, maka *i* adalah faktor dari *n*.

```
CariFaktor(input n : integer)
Deklarasi
  k : integer

Algoritma:
  k = 1
  for k = 2 to n - 1 do
    if n mod k = 0 then
      write(k)
    endif
  end for
```

Berapa Kompleksitasnya ?

5. Bilangan Prima

- Persoalan: Diberikan sebuah bilangan bilangan bulat positif. Ujilah apakah bilangan tersebut merupakan bilangan prima atau bukan.
- Definisi: bilangan prima adalah bilangan yang hanya habis dibagi oleh 1 dan dirinya sendiri.
- Algoritma brute force: bagi n dengan 2 sampai $\sqrt{n}$.
- Jika semuanya tidak habis membagi n, maka n adalah bilangan prima.

```
function Prima(input x : integer)
Algoritma:
  if x < 2 then  return false
  else
    if x = 2 then return true
    else
      y = x
      test = true
      while (test) and (y >= 2) do
        if x mod y = 0 then  test = false
        else  y = y - 1
        endif
      endwhile
```

Berapa Kompleksitasnya ?

6. Bubble Sort

- Mulai dari elemen ke-n:
  1. Jika $s_n < s_{n-1}$, pertukarkan
  2. Jika $s_{n-1} < s_{n-2}$, pertukarkan
  …
  3. Jika $s_2 < s_1$, pertukarkan

→ 1 kali *pass*

- Ulangi lagi untuk pass ke-*i*, tetapi sampai elemen ke-*i*
- Semuanya ada *n* – 1 kali *pass*

```
procedure BubbleSort (input/output s : Array,
                      input n : integer)
Deklarasi
  i,k, temp   : integer
Algoritma:
  for i ← 1 to n - 1 do
    for k ←  n downto i + 1 do
      if s[k] < s[k-1] then
        s[k] <-> s[k-1]
      endif
    endfor
  endfor
```

Tentukan Kompleksitas algoritma ini ?

7. Selection Sort

Pass ke –1:

1. Cari elemen terbesar mulai di dalam *s*[1..*n*]
2. Letakkan elemen terbesar pada posisi n (pertukaran)

P*ass* ke-2:

1. Cari elemen terbesar mulai di dalam *s*[1..*n* - 1]
2. Letakkan elemen terbesar pada posisi *n* - 1 (pertukaran)

Ulangi sampai hanya tersisa 1 elemen

Semuanya ada *n* –1 kali *pass*

```
procedure PengurutanSeleksi(input/output  s :
array)
Deklarasi
i, j, imaks, temp : integer

Algoritma:
for i <- n downto 2 do
    imaks <- 1
  for j <- 2 to i do
    if s[j] > s[imaks] then
      imaks <- j
    endif
  endfor
  s[imaks] <-> s[i]
endfor
```

8. Polinom

- Persoalan: Hitung nilai polinom

  $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

  untuk $x = t$.

- Algoritma *brute force*: x^i dihitung secara *brute force* (seperti perhitungan a^n). Kalikan nilai x^i dengan a_i, lalu jumlahkan dengan suku-suku lainnya.

8. Polinom (Perbandingan)

```
function polinom(input t : real) <- real
Deklarasi
  i, j : integer
  p, pangkat  : real

Algoritma:
  p <- 0
  for i <- n downto 0 do
    pangkat <- 1
    for j <- 1 to i do
      pangkat <- pangkat * t
    endfor
    p <- p + a[i] * pangkat
  endfor
  return p
```

VS

```
function polinom2(input x0 : real) <- real
Deklarasi
  i, j : integer
  p, pangkat  : real

Algoritma:
  p <- a[n]
  pangkat <- 1
  for i <- 1 to n do
    pangkat <- pangkat * t
    p <- p + a[i] * pangkat
  endfor
  return p
```

Tentukan kompleksitas masing-masing algortima. Manakah yang terbaik

9. Pencocokan String

Prsoalan: Diberikan

a. a. teks (*text*), yaitu (*long*) *string* dengan panjang *n* karakter

b. *pattern*, yaitu *string* dengan panjang *m* karakter (asumsi: *m* < *n*)

Carilah lokasi pertama di dalam teks yang bersesuaian dengan *pattern*.

Algoritma *brute force*:

1. Mula-mula *pattern* dicocokkan pada awal teks.
2. Dengan bergerak dari kiri ke kanan, bandingkan setiap karakter di dalam *pattern* dengan karakter yang bersesuaian di dalam teks sampai:
   - semua karakter yang dibandingkan cocok atau sama (pencarian berhasil), atau
   - dijumpai sebuah ketidakcocokan karakter (pencarian belum berhasil)
3. Bila *pattern* belum ditemukan kecocokannya dan teks belum habis, geser *pattern* satu karakter ke kanan dan ulangi langkah 2.

9. Pencocokan String

Contoh 1:

***Pattern*: NOT**

Teks: NOBODY NOTICED HIM

```
  NOBODY NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT
```

Contoh 2:

***Pattern*: 001011**

Teks: 10010101001011110101010001

```
  10010101001011110101010001
1 001011
2  001011
3   001011
4    001011
5     001011
6      001011
7       001011
8        001011
9         001011
```

9. Pencocokan String

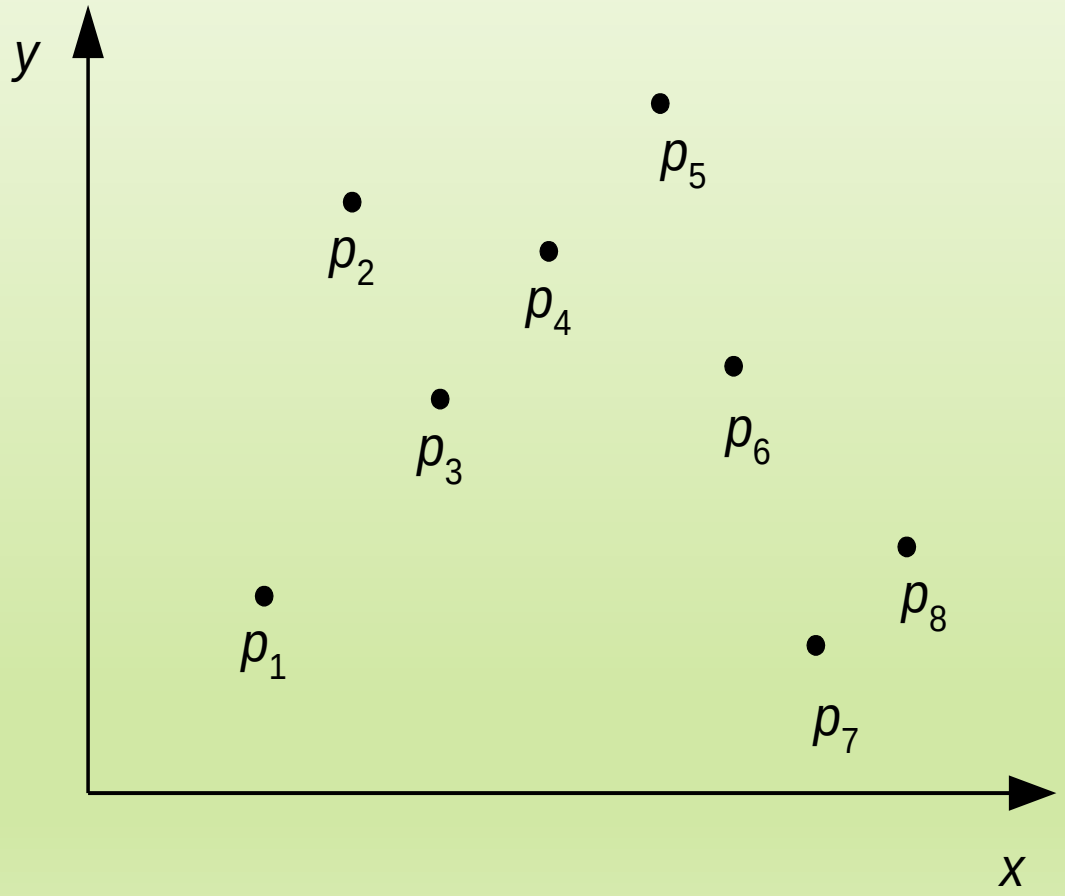
- **Kompleksitas algoritma: $O(nm)$ pada kasus terburuk**
- **$O(n)$ pada kasus rata-rata**

```
procedure PencocokanString(input P : string, T : string,
              n, m : integer, output idx : integer)
Deklarasi
  i : integer
  ketemu : boolean
Algoritma:
  i <- 0
  ketemu <- false
  while (i <=  n-m) and (not ketemu) do
    j <- 1
    while (j <=  m) and (Pj = Ti+j ) do
      j <- j+1
    endwhile
    if j = m then ketemu <- true
    else i <- i+1  {geser pattern satu karakter ke kanan teks }
    endif
  endfor
  if ketemu then idx <- i+1
  else idx <- -1
  endif
```

10. Closest Pairs

- **Persoalan: Diberikan *n* buah titik (2-D atau 3-D), tentukan dua buah titik yang terdekat satu sama lain.**

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$



10. Closest Pairs

Algoritma *brute force*:

1. **Hitung jarak setiap pasang titik.**
2. **Pasangan titik yang mempunyai jarak terpendek itulah jawabannya.**

Algoritma *brute force* akan menghitung sebanyak

$$C(n, 2) = n(n - 1)/2$$

pasangan titik dan memilih pasangan titik yang mempunyai jarak terkecil.

Kompleksitas algoritma adalah $O(n^2)$.

```
procedure CariDuaTitikTerdekat(input P : himTitik,
                               n : integer,
                               output P1, P2 : Titik)
Deklarasi
  d, dmin : real
  i, j : integer
Algoritma:
  dmin <- 9999
  for i <- 1 to n-1 do
    for j <- i+1 to n do
      d<- akar((Pi.x-Pj.x)2 + ((Pi.y-Pj.y)2)
      if d < dmin then
        dmin <- d
        P1 <-  Pi
        P2 <- Pj
      endif
    endfor
  endfor
```

Exhaustive Search (ES)

1. teknik pencarian solusi secara solusi brute force untuk masalah-masalah kombinatorik;
2. biasanya di antara objek-objek kombinatorik seperti permutasi, kombinasi, atau himpunan bagian dari sebuah himpunan.

Langkah-langkah metode exhaustive search:

1. Enumerasi (list) setiap solusi yang mungkin dengan cara yang sistematis.
2. Evaluasi setiap kemungkinan solusi satu per satu, simpan solusi terbaik yang ditemukan sampai sejauh ini (the best solusi found so far).
3. Bila pencarian berakhir, umumkan solusi terbaik (the winner)

Algoritma *exhaustive* secara teoritis menghasilkan solusi asalkan tersedia waktu atau sumberdaya yang cukup

1. Travelling Salesperson Problem (TSP)

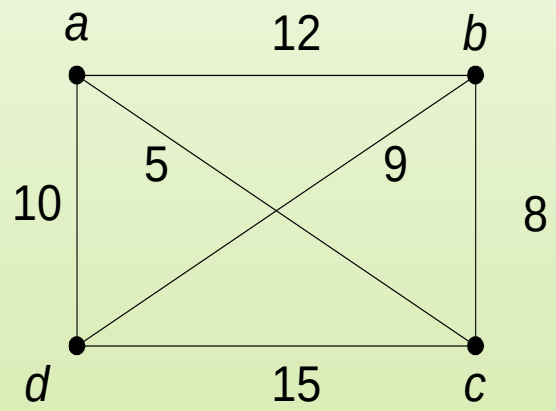
Persoalan: Diberikan *n* buah kota serta diketahui jarak antara setiap kota satu sama lain. Temukan perjalanan (*tour*) terpendek yang melalui setiap kota lainnya hanya sekali dan kembali lagi ke kota asal keberangkatan.

Persoalan TSP adalah menemukan sirkuit Hamilton dengan bobot minimum.

Algoritma exhaustive search untuk TSP:

1. **Enumerasikan (list) semua sirkuit Hamilton dari graf lengkap dengan n buah simpul.**
2. **Hitung (evaluasi) bobot setiap sirkuit Hamilton yang ditemukan pada langkah 1.**
3. **Pilih sirkuit Hamilton yang mempunyai bobot terkecil.**

1. Travelling Salesperson Problem



No.	Rute perjalanan (*tour*)	Bobot
1.	$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	10+12+8+15 = 45
2.	$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	12+5+9+15 = 41
3.	$\mathbf{a \rightarrow c \rightarrow b \rightarrow d \rightarrow a}$	10+5+9+8 = **32**
4.	$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	12+5+9+15 = 41
5.	$\mathbf{a \rightarrow d \rightarrow b \rightarrow c \rightarrow a}$	10+5+9+8 = **32**
6	$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	10+12+8+15 = 45

Rute perjalananan terpendek adalah

$\mathbf{a \rightarrow c \rightarrow b \rightarrow d \rightarrow a}$

$\mathbf{a \rightarrow d \rightarrow b \rightarrow c \rightarrow a}$

dengan bobot = 32.

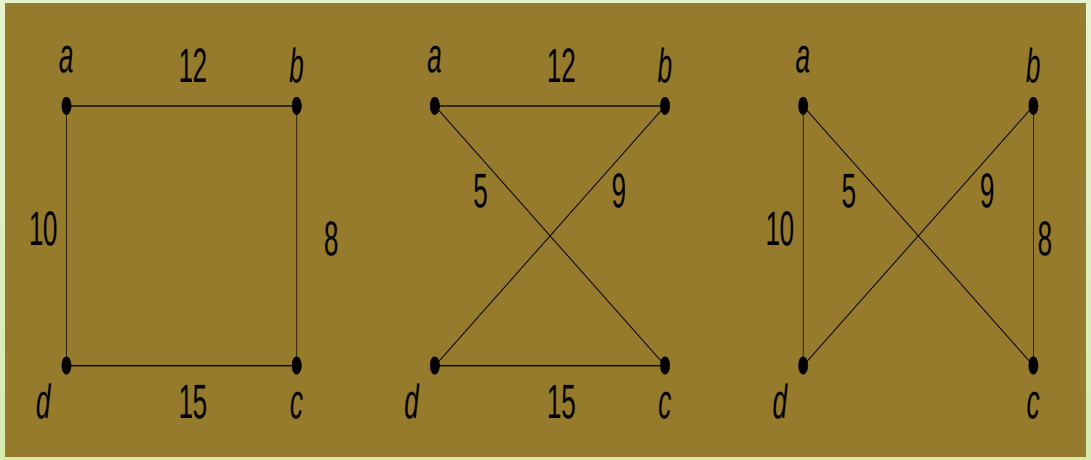
- **Untuk *n* buah simpul semua rute perjalanan dibangkitkan dengan permutasi dari *n* – 1 buah simpul.**
  - ✓ **Permutasi dari *n* – 1 buah smpul = (*n* – 1)!**
  - ✓ **Untuk *n* = 4, diperoleh (4 – 1)! = 3! = 6**

Jadi setiap sirkuit Hamilton membutuhkan waktu *O*(*n*), sehingga kompleksitas waktu algoritma *exhaustive search* untuk persoalan TSP adalah *O*(*n* · *n*!).

1. TSP (Exhaustive Search)

- **Perbaikan: setengah dari rute perjalanan adalah hasil pencerminan dari setengah rute yang lain, yakni dengan mengubah arah rute perjalanan**
  - **1 dan 6**
  - **2 dan 4**
  - **3 dan 5**
- **maka dapat dihilangkan setengah dari jumlah permutasi (dari 6 menjadi 3).**

- **Ketiga buah sirkuit Hamilton yang dihasilkan:**



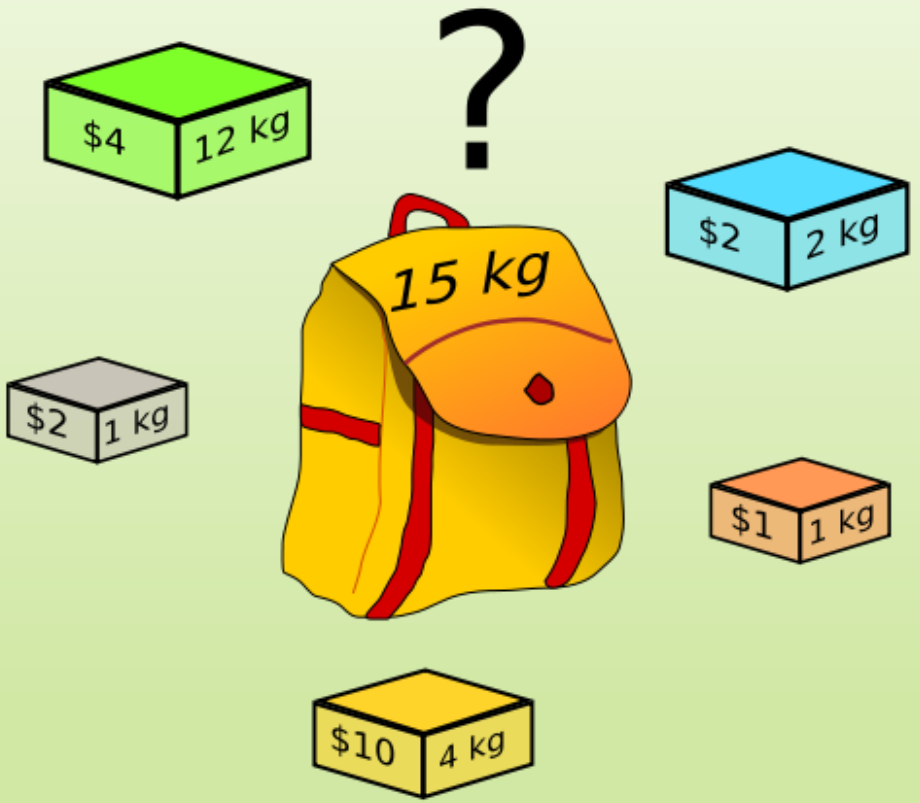
1. TSP (Exhaustive Search)

- Untuk graf dengan *n* buah simpul, kita hanya perlu mengevaluasi (*n* – 1)!/2 sirkuit Hamilton.
- Untuk ukuran masukan yang besar, jelas algoritma *exhaustive search* menjadi sangat tidak effective.
- Pada persoalan *TSP*, untuk *n* = 20 akan terdapat (19!)/2 = 6 × 1016 sirkuit Hamilton yang harus dievaluasi satu per satu.
- Sayangnya, untuk persoalan TSP tidak ada algoritma lain yang lebih baik daripada algoritma *exhaustive search*.
- Jika anda dapat menemukan algoritma yang effective untuk TSP, anda akan menjadi terkenal dan kaya!
- Algoritma yang effective selalu mempunyai kompleksitas waktu dalam orde polinomial.

2. 1/0 Knapsack (ES)

- **Persoalan: Diberikan *n* buah objek dan sebuah *knapsack* dengan kapasitas bobot *K*. Setiap objek memiliki properti bobot (*weigth*) w_i dan keuntungan(*profit*) p_i.**

Bagaimana memilih memilih objek-objek yang dimasukkan ke dalam *knapsack* sedemikian sehingga memaksimumkan keuntungan. Total bobot objek yang dimasukkan ke dalam *knapsack* tidak boleh melebihi kapasitas *knapsack*.



2. 1/0 Knapsack (ES)

- **Solusi persoalan dinyatakan sebagai:**

  $X = \{x_1, x_2, \ldots, x_n\}$

  x_i = 1, jika objek ke-*i* dipilih,

  x_i = 0, jika objek ke-*i* tidak dipilih.

Maksimasi $F = \sum_{i=1}^{n} p_i x_i$

dengan kendala (*constraint*)

$$\sum_{i=1}^{n} w_i x_i \le K$$

yang dalam hal ini, $x_i = 0$ atau 1, $i = 1, 2, \ldots, n$

2. 1/0 Knapsack (ES)

Algoritma *exhaustive search:*

1. **Enumerasikan (*list*) semua himpunan bagian dari himpunan dengan n objek.**

2. **Hitung (evaluasi) total keuntungan dari setiap himpunan bagian dari langkah 1.**

3. **Pilih himpunan bagian yang memberikan total keuntungan terbesar.**

Contoh: $n = 4$.

$w_1 = 2$; $p_1 = 20$

$w_2 = 5$; $p_2 = 30$

$w_3 = 10$; $p_3 = 50$

$w_4 = 5$; $p_4 = 10$

Kapasitas *knapsack* $K = 16$

Langkah-langkah pencarian solusi 0/1 *Knapsack* secara *exhaustive search* dirangkum dalam tabel di bawah ini:

2. 1/0 Knapsack (ES)

- Berapa banyak himpunan bagian dari sebuah himpunan dengan *n* elemen? Jawabnya adalah 2^n.

  Waktu untuk menghitung total bobot objek yang dipilih = $O(n)$

  Sehingga, Kompleksitas algoritma *exhaustive search* untuk persoalan 0/1 *Knapsack* = $O(n.\ 2^n)$.

- TSP dan 0/1 *Knapsack*, adalah contoh persoalan eksponensial. Keduanya digolongkan sebagai persoalan *NP* (*Non-deterministic Polynomial*), karena tidak mungkin dapat ditemukan algoritma polinomial untuk memecahkannya.

- Himpunan bagian objek yang memberikan keuntungan maksimum
- adalah {2, 3} dengan total keuntungan adalah 80.
- Solusi: X = {0, 1, 1, 0}

- ???

Latihan 1 (selesaikan secara *exhaustive search*)

(Masalah Penugasan) Misalkan terdapat *n* orang dan *n* buah pekerjaan (*job*). Setiap orang akan di-*assign* dengan sebuah pekerjaan. Penugasan orang ke-*i* dengan pekerjaan ke-*j* membutuhkan biaya sebesar *c*(*i*, *j*). Bagaimana melakukan penugasan sehingga total biaya penugasan adalah seminimal mungkin? Misalkan instansiasi persoalan dinyatakan sebagai matriks *C* sebagai berikut

$$C = \begin{bmatrix} Job\,1 & Job\,2 & Job\,3 & Job\,4 & \\ 9 & 2 & 7 & 8 & \text{Orang } a \\ 6 & 4 & 3 & 7 & \text{Orang } b \\ 5 & 8 & 1 & 4 & \text{Orang } c \\ 7 & 6 & 9 & 4 & \text{Orang } d \end{bmatrix}$$

Latihan 2

(Masalah partisi). Diberikan *n* buah bilangan bulat positif. Bagilah menjadi dua himpunan bagian *disjoint* sehingga setiap bagian mempunyai jumlah nilai yang sama (catatan: masalah ini tidak selalu mempunyai solusi).

Contoh: *n* = 6, yaitu 3, 8, 4, 6, 1, 2, dibagidua menjadi {3, 8, 1} dan {4, 6, 2} yang masing-masing jumlahnya 12.

Rancang algoritma *exhaustive search* untuk masalah ini. Cobalah mengurangi jumlah himpunan bagian yang perlu dibangkitkan.

Latihan 3

(Bujursangkar ajaib). Bujursangkar ajaib (*magic square*) adalah pengaturan *n* buah bilangan dari 1 hingga n^2 di dalam bujursangkar yang berukuran *n* x *n* sedemikian sehingga jumlah nilai setiap kolom,baris, dan diaginal sama. Rancanglah algoritma *exhaustive search* untuk membangkitkan bujursangkar ajaib orde *n*.

4	9	2
3	5	7
8	1	6

Kriptografi (Exhaustive Search)

- Di dalam kriptografi, *exhaustive search* merupakan teknik yang digunakan penyerang untuk menemukan kunci enkripsi dengan cara mencoba semua kemungkinan kunci.

  Serangan semacam ini dikenal dengan nama *exhaustive key search attack* atau *brute force attack*.

- Contoh: Panjang kunci enkripsi pada algoritma *DES* (*Data Encryption Standard*) = 64 bit.
- Dari 64 bit tersebut, hanya 56 bit yang digunakan (8 bit paritas lainnya tidak dipakai).
- Jumlah kombinasi kunci yang harus dievaluasi oleh pihak lawan adalah sebanyak

  $(2)(2)(2)(2)(2) \ldots (2)(2) = 2^{56} = 7.205.759.403.7927.936$
- Jika untuk percobaan dengan satu kunci memerlukan waktu 1 detik, maka untuk jumlah kunci sebanyak itu diperlukan waktu komputasi kurang lebih selama 228.4931.317 tahun!

Algoritma Heuristic

- **Algoritma exhaustive search tidak effective sebagaimana ciri algoritma brute force pada umumnya**
- **Namun, nilai plusnya terletak pada keberhasilannya yang selalu menemukan solusi (jika diberikan waktu yang cukup).**
- **Algoritma exhaustive search dapat diperbaiki kinerjanya sehingga tidak perlu melakukan pencarian terhadap semua kemungkinan solusi.**
-
- **Salah satu teknik yang digunakan untuk mempercepat pencarian solusi, di mana exhaustive search tidak praktis, adalah teknik heuristik (heuristic).**
- **Dalam exhaustive search, teknik heuristik digunakan untuk mengeliminasi beberapa kemungkinan solusi tanpa harus mengeksplorasinya secara penuh.**

Algoritma Heuristic

- **Heuristik adalah teknik yang dirancang untuk memecahkan persoalan dengan mengabaikan apakah solusi dapat terbukti benar secara matematis**
- **Contoh dari teknik ini termasuk menggunakan tebakan, penilaian intuitif, atau akal sehat.**
- **Contoh: program antivirus menggunakan pola-pola heuristik untuk mengidentifikasi dokumen yang terkena virus atau *malware*.**

Sejarah

- **Heuristik adalah seni dan ilmu menemukan (*art and science of discovery*).**

Kata heuristik diturunkan dari Bahasa Yunani yaitu "*eureka*" yang berarti "menemukan" (*to find* atau *to discover*).

Algoritma Heuristic

- **Heuristik berbeda dari algoritma:**

  - heuristik berlaku sebagai panduan (*guideline*),

  - sedangkan algoritma adalah urutan langkah-langkah penyelesaian masalah.

- **Heuristik mungkin tidak selalu memberikan hasil yang diinginkan, tetapi secara ekstrim ia berguna pada pemecahan masalah.**

- **Heuristik yang bagus dapat secara dramatis mengurangi waktu yang dibutuhkan untuk memecahkan masalah dengan cara mengeliminir kebutuhan untuk mempertimbangkan kemungkinan solusi yang tidak perlu.**

Matematikawan Yunani yang bernama Archimedes yang melontarkan kata "*heureka*", dari sinilah kita menemukan kata "*eureka*" yang berarti "*I have found it*."

Algoritma Heuristic

- **Heuristik tidak menjamin selalu dapat memecahkan persoalan, tetapi seringkali memecahkan persoalan dengan cukup baik untuk kebanyakan persoalan, dan seringkali pula lebih cepat daripada pencarian solusi secara *exhaustive search*.**

- **Sudah sejak lama heuristik digunakan secara intensif di dalam bidang intelijensia buatan (*artificial intelligence*).**

Algoritma Heuristic

- Algoritma Heuristic heuristik untuk mempercepat algoritma exhaustive search
- Contoh: Masalah anagram.
- Anagram adalah penukaran huruf dalam sebuah kata atau kalimat sehingga kata atau kalimat yang baru mempunyai arti lain.

Contoh-contoh anagram (semua contoh dalam Bahasa Inggris):

lived → devil

tea → eat

charm → march

- Bila diselesaikan secara *exhaustive search*, kita harus mencari semua permutasi huruf-huruf pembentuk kata atau kalimat, lalu memerika apakah kata atau kalimat yang terbentuk mengandung arti.
- Teknik heuristik dapat digunakan untuk mengurangi jumlah pencarian solusi. Salah satu teknik heuristik yang digunakan misalnya membuat aturan bahwa dalam Bahasa Inggris huruf *c* dan *h* selalu digunakan berdampingan sebagai *ch* (lihat contoh *charm* dan *march*), sehingga kita hanya membuat permutasi huruf-huruf dengan *c* dan *h* berdampingan. Semua permutasi dengan huruf *c* dan *h* tidak berdampingan ditolak dari pencarian.

Ada Pertanyaan ??